Aventura!

Gubi

6 de setembro de 2018

Sumário

1 Introdução **1**

2 Estrutura interna **2**
2.1 Elementos 3
2.1.1 Lugares 3
2.1.2 Objetos 3
2.2 Saídas 4
2.3 Verbo 4
2.4 O aventureiro 4

3 Biblioteca básica **4**
3.0.1 Tabela de símbolos 5
3.0.2 Lista de valores 5
3.1 Programa da primeira fase 5
3.2 Sugestões 6

1 Introdução

O objetivo do projeto é montar um sistema de execução de jogos do tipo *adventure*, como apresentado em classe. Neste tipo de jogo o personagem principal (aventureiro) executa ordens dadas pelo jogador e descreve o que aconteceu, sempre em forma de texto.

A estória acontece em um mundo virtual composto de lugares interligados por passagens, transferências mágicas, teletransporte, etc. Neste mundo existem vários objetos (virtuais, lógico) espalhados e que podem interagir com o aventureiro de diversas formas (veja a seção 2.1.2).

Existem alguns jogos disponíveis em diversos lugares:

- No emacs, execute `M-x dunnet`

- No linux, rode o programa `adventure`, que nas distribuições *Ubuntu* e *Debian* fica no pacote `bsdgames.deb`. Este jogo é uma adaptação do primeiro escrito neste estilo, o "Colossal Cave", em FORTRAN!
- Outra possibilidade com o linux é o `battlestar`, também em `bsdgames.deb`
- Procure jogos da *Infocom*, que marcou época com seus jogos de texto interativo. No linux existem vários pacotes para interpretar estes jogos. Experimente o `frotz` e o `gargoyle-free`.
- Experimente o engraçadíssimo *The Hitchhikers Guide to the Galaxy*. Passarei mais detalhes na sala de aula.

O projeto será desenvolvido em C e terá as seguintes componentes fundamentais:

- Biblioteca com os elementos básicos para a construção do jogo.
- Mecanismo interno do funcionamento do jogo.
- Interface elementar com o usuário.
- Interpretador da linguagem (quase) natural e finalização.

Além disso, o grupo deverá montar uma estória, seguindo rigidamente a especificação adotada no curso. Estórias montadas por um grupo devem poder ser usadas por outro grupo sem a necessidade de uma adaptação.

2 Estrutura interna

Antes de prosseguir, vou adiantar um pouco da estrutura interna a ser usada na descrição do mundo e seus objetos. Mais adiante, após termos visto melhor a noção de objetos e herança, retornarei a este ponto e o aprofundarei.

O mundo é composto por **elemento**s, que por sua vez podem ser lugares ou objetos. Os elementos serão implementados por *struct*s em C, usando modularização. Usarei **objeto** para um objeto dentro do mundo virtual. Para ajudar a fixar, **lugar** indicará uma região do mundo virtual.

Além dos **elemento**s, a descrição do jogo ainda contém **saída**s e **verbo**s. As **saída**s são conexões entre **lugar**es e os **verbo**s descrevem as ações possíveis.

A **objeto** e **lugar** são derivações de **elemento**, já que sua estrutura é parecida, mas existem algumas diferenças que estão discutidas nas próximas seções. Todo **elemento** possui algumas propriedades que o descrevem, ou a seu estado, e pode ser alvo de um conjunto de ações. Os **elemento**s devem conter um nome interno, um ou mais apelidos, uma descrição longa e uma descrição curta.

Os apelidos indicarão como o usuário irá se referenciar ao elemento. Por exemplo um *struct* que descreve um gato pode ter nome "gt" e apelidos "gato", "felino" e "bichano". funcionar como apelido também.

Nesta fase trataremos os **lugar**es e os *struct*s de forma independente, embora em java eles sejam relacionados. Assim, no que segue, haverá alguma redundância.

2.1 Elementos

Tanto **objeto**s como **lugar**es possuem as seguintes propriedades. O termo entre parênteses é o que será utilizado na especificação para indicar a propriedade.

- Nome — identificação interna do **elemento**, deve ser um nome único (representa a variável associada).
- Artigos (artigos) — uma lista de quatro palavras, indicando os artigos: definido direto, definido indireto, indefinido direto, indefinido indireto. Por exemplo: "o do um dum".
- Descrição longa (longa) — texto que descreve detalhadamente o **elemento**, apresentada quando o usuário pede para examinar o **objeto** ou **lugar**.
- Descrição curta (curta)— descrição abreviada, sem detalhes. É a descrição normalmente apresentada.
- Lista de objetos (contem) — lista dos nomes dos **objeto**s presentes no **lugar** ou contidos no **objeto**. Opcional.
- Atributos — são valores genéricos que permitem especificar melhor o estado, de acordo com as necessidades do jogo. Opcionais.
- Ação (acao) — **verbo** que tem significado especial neste **lugar** ou para este **objeto**. É opcional e pode haver mais de uma.
- Animação (animacao) — **verbo** especial que é chamado a cada iteração, permitindo animações. É opcional.

2.1.1 Lugares

Um **lugar** possui a seguinte propriedade adicional:

- Saída (saida) — conexão para outro **lugar**. As direções possíveis são pelos apelidos de cada saída (ver mais adiante).

2.1.2 Objetos

Além das propriedades dos **elemento**s, os **objeto**s possuem ainda:

- Adjetivos (adjetivos) — lista adicional com adjetivos que permitem especificar melhor o **objeto**.

- Invisível (invisivel) — indicação especial para **objeto**s que estão escondidos no início do jogo.

As açoes genéricas para **objeto**s são as seguintes:

- `examinar` — descreve o objeto.
- `pegar` — passa o objeto para o aventureiro, se possível (podem haver restrições quanto a número, tamanho ou peso que o aventureiro pode carregar).
- `largar` — o **objeto** precisa estar com o aventureiro. O **objeto** é colocado no lugar onde o aventureiro se encontra. Com argumentos adicionais, pode-se colocar um **objeto** dentro de outro, se possível.

Ações específicas para alguns objetos podem ser `destruir`, `esfregar`, `ligar`, etc. Procure ter uma implementação padrão para verbos mais gerais. Na especificação da estória, o usuário poderá definir novos verbos.

2.2 Saídas

As **saída**s ligam um **lugar** a outro. Uma **saída** é um caso especial de **objeto**, que possui duas propriedades especiais a mais:

- Destino (destino) — contém a referência para o **lugar** onde ela leva.
- Fechada (fechada) — indica que a **saída** está fechada.

2.3 Verbo

Um **verbo** é essencialmente uma função que atua sobre os **elemento**s. Veremos sua descrição nas próximas fases.

2.4 O aventureiro

O aventureiro é um caso muito especial de **objeto** animado. Ele pode conter outros **objeto**s, se os estiver carregando ou vestindo, e recebe atualizações diretamente do usuário.

Além disso, podem haver outras atualizações automáticas, como cansaço, fome e recuperação, se a especificação do jogo indicar.

3 Biblioteca básica

Para construirmos tudo isso, precisamos de algumas estruturas fundamentais: uma tabela de símbolos que liga nomes a **elemento**s e listas de diversos tipos.

3.0.1 Tabela de símbolos

A tabela de símbolos deve ser implementada por uma tabela de *hash* (detalhes em aula), onde a chave é uma *string* e o valor é um ponteiro genérico.

Além da estrutura de dados pura, devem ser implementadas as seguintes funções:

- `TabSim cria(int tam)` — cria uma tabela com `tam` entradas.
- `void destroi(TabSim t)` — destroi a tabela `t`.
- `int insere(TabSim t, char *n, Elemento *val)` — insere o nome `n` na tabela `t` e o associa com o valor `val`. Deve retornar um código de erro, indicando se a inserção foi bem sucedida ou não.
- `Elemento *busca(TabSim t, char *n)` — retorna o valor associado a `n` na tabela `t`. Deve retornar `NULL` caso o valor não seja encontrado.
- `int retira(TabSim t, char *n)` — remove o valor `n` da tabela, liberando a memória.

3.0.2 Lista de valores

As listas devem ser implementadas como listas ligadas (detalhes em aula). Da mesma forma que a tabela de símbolos, a implementação deve conter um conjunto de funções:

- `Lista cria()` — cria uma lista vazia.
- `void destroi(Lista l)` — destroi a lista `l`.
- `Lista insere(Lista l, Elemento *val)` — insere o valor `val` na lista `l` retornando o endereço do elemento inserido, ou `NULL` em caso de erro. Veja em aula a razão de fazermos desta forma.
- `Elemento *busca(Lista l, char *n)` — retorna o valor associado ao nome `n` na lista `l`. Deve retornar `NULL` caso o elemento não seja encontrado.
- `Elemento *retira(Lista l, Elemento *val)` — remove o elemento `*val` da tabela, sem removê-lo da memória.

3.1 Programa da primeira fase

O programa da primeira fase deve conter os módulos com as estruturas acima e um modulo adicional para testes. Todas as funções devem ser testadas em várias situações, para garantir que a biblioteca não contém erros.

Junto com o programa, deve também ser entregue um relatório simples com instruções de como usar o programa, detalhes da implementação e lista de eventuais problemas encontrados.

3.2 Sugestões

Para esta primeira fase, defina uma *struct* para elementos, com informações mínimas:

```
typedef struct {
  char n[80];
} Elemento;
```

Nas próximas fases, colocaremos o resto das informações.

Para a lista ligada, é muito mais prático ter uma estrutura separada com um ponteiro para a cabeça da lista. Desta forma, o endereço da lista não muda com inserções e deleções:

```
#include "elemento.h"

typedef struct elo {
  struct elo *next;
  Elemento *val;
} Elo;

typedef struct {
  Elo *cabec;
} Lista;
```

Para não tirar toda a diversão, deixo a tabela de hash inteiramente por conta de vocês. Há documentação *online* no próprio IME.